Matjaž B. Jurič
Marjan Heričko
Ivan Rozman
József Györkös
University of Maribor, Faculty of Electrical Engineering
and Computer Science, Institute of Informatics
Maribor, Slovenia
E-mail: matjaz.juric@uni-mb.si

UDC: 681.32.06
Review Article

Software Reuse by Using Components

Object technology is evolving into component based industry. This paper shows how to achieve software reuse by using component technology. Basic ideas of component technology are shown and two well-known models, CORBA and DCOM, are described. It ialso shows how distributed object models support interface and implementation reuse. The advantages and disadvantages are indicated too. Component technology is mature enough and ready for practical use and there is a huge potential of component technology to offer substantial benefits in software development process.

Keywords: reuse, distributed object, component, CORBA, DCOM.

1. Introduction

The vision of building new systems out of reusable software components rather than creating each new application from scratch is so compelling that hardly anyone questions the value of reuse. Reuse is often mentioned as a feature of object technology, universally regarded as an inherent good. Object technology is believed to be crucial to achieve the long-sought after goal of widespread reuse.

Unfortunately, many people naively equate reuse with objects, adopting it to »automatically« ensure reuse, but often do not get much reuse. There are also many examples of successful reuse using non-object oriented languages (function libraries for example). Success of these libraries depends on the programmer's knowledge and motivation for searching and using the code.

Thus objects are neither necessary nor sufficient for effective reuse. Without an explicit reuse agenda, object oriented reuse will not succeed. In almost all cases of successful reuse, non-technical issues such as management support and a stable domain seem to dominate over specific language or design methodology.

2. Components and reuse

The concept of component is still evolving, although the general thought is that most software artifacts can be considered as components. Definitions of a component are:

- A component is a subsystem, use case, actor, or any object class;
- A component is a good abstraction for higher-level design, with access restricted by visibility rules;
- A component is not bound to any specific application;
- A component is a high-quality product due to careful design and testing;
- A component is packaged for reuse with well-designed interface, documentation;
- A component is general so that it can be used in several places;
- A component is specialized when used.

This emphasizes components as high-quality, generic software products with well-defined public interfaces, designed to be used in many contexts.

Another possible, though much simpler, definition of a component is: A component is a package of functionality, deployed within specific technology framework [1]. Here, »package of functionality« refers to a high-level, reusable abstraction with one or more public interfaces. This emphasizes that a component is not necessarily restricted to object technology. In fact, components have been in existence for decades in the form of high-quality libraries of routines and functions.

A reusable component is any component, developed for reuse and actually used in more than one context [5]. Reusable components can be code, design specifications, processes, methods, documentation, system or subsystems, models, patterns, frameworks, classes, object implementations etc.

Before something can be reused, we must assure that it is ready for reuse. Before the reuse is possible, we have to:

- locate the component,
- know, what the component is doing and
- know, how to reuse the component.

It this article the way software code can be reused will be discussed. There are two ways, the code can be reused:

- source code reuse and
- binary reuse.

Source code reuse has been known for-long time. In object oriented programming languages it is achieved through inheritance. Binary reuse is extremely difficult to achieve and hasn't been used recently.

3. Components or distributed objects

In developing computer software, object-oriented programming offers a model different from traditional structured programming and design, which is based on functions and procedures [3]. In simplified terms, object-oriented programming is a way to develop software by building self-contained modules, that can be more easily replaced, modified and reused.

The proponents of object-oriented programming (OOP) promised that this new style of programming would help solve all application deployment problems. In fact, C++, the most widely used OOP language, has proved to be the language-of-choice for independent software vendors developing commercial applications, and also for the architectural components of many corporate information systems. On the other hand, OOP languages have not achieved all of their promised benefits, and the vision of a world of reusable, interchangeable business objects to aid in software development and system deployment has not been achieved.

A »classical« object - of the C++ or Smalltalk variety - is a blob of intelligence that encapsulates code and data. However, classical objects exist only within a single program. Only the language compiler that creates the objects knows of their existence [4]. The outside world does not know about these objects and has no way to access them.

The OOP languages have merely defined a way to specify the internal details for a component, they have not provided a standard interface for these components to connect. As a result, all OOP objects are dependent on the implementation of other objects (that is, they can only connect to components that they were specifically designed to work with).

In contrast, a *distributed object* is a blob of intelligence that can live anywhere on a network. Distributed objects are packaged as independent pieces of code that can be accessed by remote clients via method invocations. The programming language and compiler used to create distributed objects are totally transparent. Clients do not need to know, where the distributed object resides or what operating system it executes on. Distributed objects are intelligent pieces of software that can message each other transparently.

When we talk about distributed objects, we are really talking about independent software *components*. A component can be seen as an object, that is not bound to a particular program, computer language, or implementation [2]. Notice that we have been using the terms »components« and »distributed objects« interchangeably. In distributed object systems, the unit of work and distribution is a component.

4. Distributed object models

To make distributed objects reality a distributed object model is needed. Distributed object model is a middleware that allows objects to distribute across the

network and to communicate with each other [9]. Corporate application developers can use distributed object models to create new solutions that combine in-house business objects, off-the-shelf objects, and their own custom components.

Today several object models exist. The most important are *Common Object Request Broker Architecture* (CORBA) from *Object Management Group* (OMG) and *Distributed Component Object Model* (DCOM) from *Microsoft*.

CORBA is the most important and ambitious middleware project ever undertaken by industry [7]. It is a product of a consortium - called the *Object Management Group* - that includes over 700 companies[1], representing the entire spectrum of the computer industry. CORBA was designed to allow intelligent components to discover each other and interoperate on an object bus. However, CORBA goes beyond just interoperability (Figure 1). It also specifies an extensive set of bus-related services and common facilities.

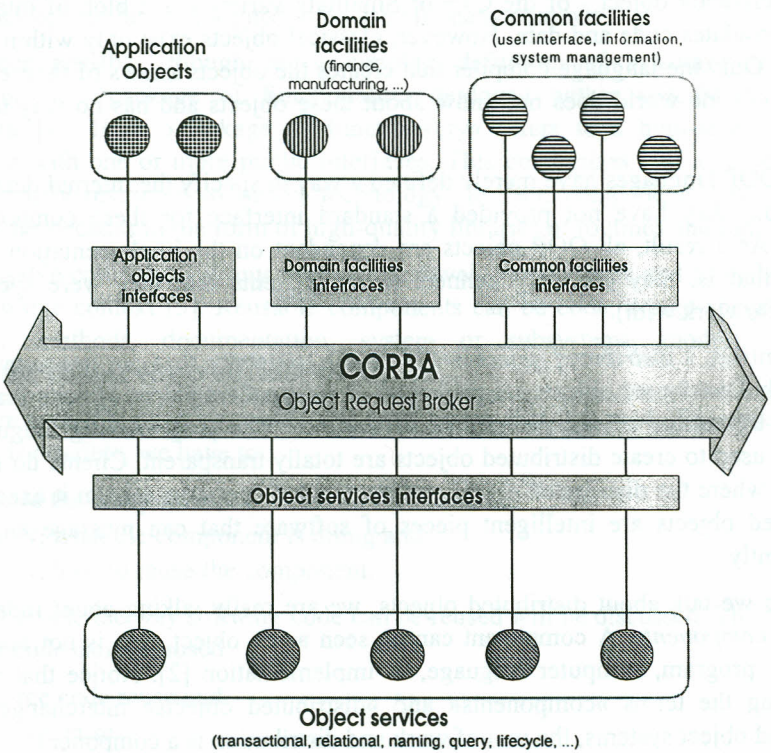

Figure 1: *CORBA architecture*

[1] University of Maribor, Faculty of Electrical Engineering and Computer Science, Institute of Informatics is member of Object Management Group (OMG) since January 1996.

DCOM is an extension of the *Component Object Model* (COM), which has been part of the Windows family of operating systems for many years as the underlying framework that makes OLE, and more recently ActiveX, possible. COM's binary interoperability standard facilitates independent development of software components and supports deployment of those components in binary form. The result is that independent software vendors can develop and package reusable building blocks without shipping source code. DCOM extends COM to the network with remote method calls, security, scalability, and location transparency (Figure 2).

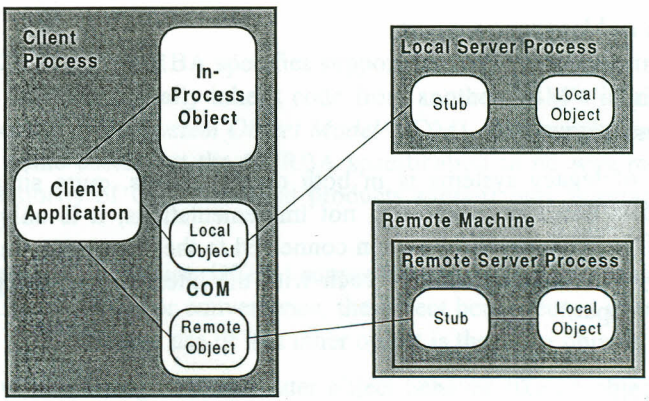

Figure 2: *DCOM provides location and packing transparency.*

Despite the ongoing debate about which technology is superior, you will find striking similarities. Both CORBA and DCOM provide **separation of an object's interface from its implementation(s)**. This separation enables much greater software reuse, by freeing the user of an object from knowledge of implementation details. Clients of an object depend only on its interface. Once the interface is defined, any number of implementations can be written to support that same interface.

Both architectures achieve this separation through the use of their own *interface definition language* (IDL). IDL is an independent programming language allowing clients and servers to be written in different languages.

Another key attribute shared by CORBA and DCOM is **location transparency**. The code to access methods of local objects is the same as that for accessing remote objects. However, under the covers, the distributed object architecture does substantial work to make remote method invocation easy for developers. To provide this transparency, both models provide marshaling code that ensures proper data representation regardless of object location. To support location and access transparency, memory management rules and routines are also provided. Each

architecture provides build-in error handling support to complete the method invocation mechanism.

Both architectures support interface inheritance and provide polymorphism when dispatching method invocation requests. CORBA and DCOM store interface definitions in a repository. Both also support dynamic method invocation. Finally, server registration and activation are integral to the two architectures.

CORBA and DCOM are similar, but there are differences. It is difficult to make an objective comparison, because there is only one implementation of DCOM (from Microsoft) and nearly a dozen different CORBA products. The differences are in following areas:

- platform and language support,
- cost,
- integration,
- robustness.

Integration of legacy systems is in both object models, quite simple. Since all communication is based on interfaces, not implementations, it is only necessary to build the interface. This interface is then connected to the legacy system, which is not necessaryly object oriented. This approach with the interface and the proxy code is called object wrapper.

5. Source code reuse

An important goal of any object model is that component authors can reuse and extend objects provided by others as pieces of their own component implementations. One way this can be achieved is implementation inheritance: to reuse code in the process of building a new object, you inherit implementation from it and override methods in the tradition of C++ and other languages. However, as a result of many years experience, many people believe traditional language-style implementation inheritance technology as the basis for object reuse is simply net robust enough for large, evolving systems composed of software components.

The problem for system-wide object interaction using traditional implementation inheritance is that the contract (the interface) between objects in an implementation hierarchy is not clearly defined. In fact, it is implicit and ambiguous. When the parent or child object changes its implementation, the behavior or related components may become undefined, or unstably implemented. In any single application, where the implementation can be managed by a single engineering team, who update all of the components at the same time, this is not always a major concern. In an environment where the components of one team are built through black-box reuse of other components built by other teams, this type of instability jeopardizes reuse.

Additionally, implementation reuse usually works only within process boundaries. This makes traditional implementation inheritance impractical for large, evolving systems composed of software components built by many engineering teams [6].

The key to building reusable components is to be able to treat the object as a black box. This means that the piece of code attempting to reuse another objects knows nothing, and needs to know nothing about the internal structure or implementation of the component being used [8]. In other words, the code attempting to reuse a component depends upon the behavior of the object and not the exact implementation.

6. Component (binary) reuse

Neither DCOM nor CORBA specifies support for implementation inheritance, the ability of one object to actually inherit code from another. IBM's implementation of CORBA, embodied in the *System Object Model* (SOM), does support implementation inheritance in some cases, but the CORBA specification in no way requires this. In fact, a great majority of CORBA-based products support only interface inheritance, just like DCOM.

To achieve black-box reuse, DCOM supports two mechanisms through which one object may reuse another. For convenience, the object being reused is called the *inner* object and the object making use of that inner object is the *outer* object [10].

1. **Containment/Delegation** the outer object behaves like an object client to the inner object. The outer object contains the inner object and when the outer object wishes to use the services of the inner object the outer object simply delegates implementation to the inner object's interfaces. In other words, the outer object uses the inner's services to implement itself. It is not necessary that the outer and inner objects support the same interfaces; in fact, the outer object may use an inner object's interface to help implement parts of a different interface on the outer object especially when the complexity of the interfaces differs greatly.
2. **Aggregation** the outer object wishes to expose interfaces from the inner object as if they were implemented on the outer object itself. This is useful when the outer object would always delegate every call to one of its interfaces to the same interface of the inner object. Aggregation is a convenience to allow the outer object to avoid extra implementation overhead in such cases.

These two mechanisms are illustrated in Figures 3 and 4. The important part to both these mechanisms is how the outer object appears to its clients. As far as the clients are concerned, both objects implement interfaces A, B, and C. Furthermore, the client treats the outer object as a black box, and thus does not care, nor does it need to care, about the internal structure of the outer object - the client only cares about behavior.

A DCOM interface defines the behavior or capabilities of a software component as a set of methods and properties. An interface is a contract that guarantees consistent semantics from objects that support it. Each DCOM object must support at least one interface called *IUnknown,* although it may support many interfaces simultaneously. *IUnknown* defines methods that provide the basic building blocks for managing object life cycles and allowing graceful evolution of interfaces supported by an object.

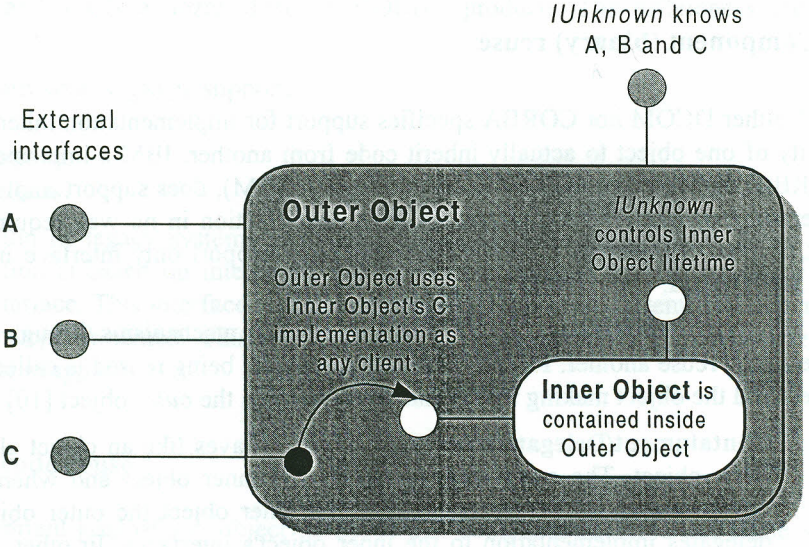

Figure 3: *Containment of an inner object and delegation to its interfaces.*

Containment is simple to implement for an outer object: during its creation, the outer object creates any inner objects it needs to use as any other client would. This is nothing new - the process is like a C++ object that itself contains a C++ string object that it uses to perform certain string functions even if the outer object is not considered a string object in its own right.

Aggregation is almost as simple to implement, the primary difference being the implementation of the three *IUnknown* functions: *QueryInterface*, *AddRef,* and *Release*. The catch is that from the client's perspective, any *IUnknown* function on the outer object must affect the outer object. That is, *AddRef* and *Release* affect the outer object and *QueryInterface* exposes all the interfaces available on the outer object. However, if the outer object simply exposes an inner object's interface as it's own, that inner object's *IUnknown* members called through that interface will behave differently than those *IUnknown* members on the outer object's interfaces, a sheer violation of the rules and properties governing *IUnknown.*

The solution is for the outer object to somehow pass the inner object some *IUnknown* pointer to which the inner object can re-route (that is, delegate) *IUnknown* calls in its own interfaces, and yet there must be a method through which the outer object can access the inner object's *IUnknown* functions that only affect the inner object.

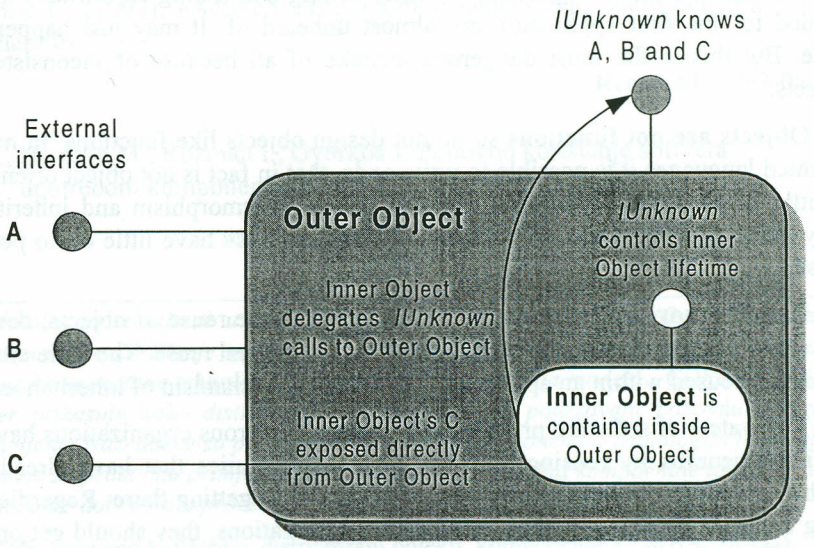

Figure 4: *Aggregation of an inner object where the outer object exposes one or more of the inner object's interfaces as its own.*

7. Reuse - just another software fad?

Let's reconsider software reuse in the light of business rather than technical goals. There are three major benefits when reuse is properly managed:

- It helps a company develop new applications faster, responding more quickly to changes in the business environment.
- It provides a shared view of the business through development of common objects to represent customer, products, services, and other key business components.
- It minimizes the total amount of code, that must be maintained by a company.

These are powerful benefits. They provide good reasons for investing in reusable components. Unfortunately, only few companies recognize these benefits and make

the investment required to achieve them. Most companies make one or more of the following mistakes in pursuing reuse within their object-oriented development efforts.

Reuse is not free. An object designed for one application is not likely to satisfy the needs of other applications unless a special effort is made by developers to ensure reuse. Such an effort consumes considerable time and resources, slowing down the development process.

Do not focus on code reuse. It is rare to see the requisite analysis and design go into an object to make it generic - actually coding and testing capabilities that are not needed for the first application are almost unheard of. It may just happen to reuse code. But this is the most dangerous mistake of all because of inconsistent use of objects.

Objects are not functions so do not design objects like functions. In most object oriented languages it is possible to write code, that in fact is not object oriented. There is little or no object autonomy, minimal use of polymorphism and inheritance, and only the loosest form of encapsulation. These »objects« have little or no potential for reuse.

Do not ignore internal reuse. As powerful as the reuse of objects, developed in one application and reused in new application, is internal reuse. The code embedded in objects is reused within an application through the mechanism of inheritance.

The waters of software problems are rising. Numerous organizations have attained business benefits by reusing software assets. Companies that have already reached higher ground know that software reuse is critical to getting there. Regardless of how long the road to managed reuse will be for organizations, they should get on that road now. They should work on maturing its development process and adopt object based reuse techniques. Reuse efforts will payoff only in conjunction with effective frameworks. And remember, a component is not reuse.

References:

[1] Baster G. (1997), »Business Components for End-User Assembly«, Object Magazine, Vol 6(11)

[2] Guttman M., Matthews J. R. (1995), »The Object Technology Revolution«, John Wiley & Sons, Inc., USA

[3] Jurič M. B., Heričko M., Rozman I. (1997), »Objektni modeli porazdeljenega procesiranja«, Dnevi slovenske informatike, Portorož 1997

[4] Jurič M. B., Heričko M., Rozman I. (1997), »CORBA - objektni model porazdeljenega procesiranja«, Uporabna informatika, jan/feb/mar 1997

[5] Krmac Vatovec E. (1997), »Ponovna uporaba - kaj, zakaj, kdaj in kako«, Dnevi slovenske informatike, Portorož 1997

[6] McGregor D. J., Doble J., Keddy A. (1996), »A Pattern for Reuse«, Object Magazine, Vol. 6(2)

[7] OMG (1995), »The Common Object Request Broker: Architecture and Specification«, Revision 2.0

[8] Rajeev V. (1997), »Reusing Business Objects«, Object Magazine, Vol. 6(11)

[9] Soley R. M., Ph.D. (1995), »Object Management Architecture Guide«, OMG, John Wiley & Sons, Inc., Revision 3.0, Third Edition

[10] Weiying C. (1997), »ActiveX Programming Unleashed«, Sams.net Publishing, Indianapolis

Received: 1997-08-15

Jurič M. B., Heričko M., Rozman I., Györkös J. Ponovno korištenje softvera upotrebom komponenata

Sažetak

Objektna tehnologija razvija se u industriju temeljenu na komponentama. Ovaj rad pokazuje kako postići ponovno korištenje softvera upotrebom komponentne tehnologije. Prikazane su osnovne ideje komponentne tehnologije, kao i dva dobro poznata modela - CORBA i DCOM. Rad također prikazuje kako distribuirani objektni modeli podržavaju ponovnu upotrebu sučelja i primjenu. Naznačene su prednosti i nedostaci. Komponentna tehnologija je dovoljno zrela i spremna za praktičnu primjenu. Postoji ogromni potencijal kompoentne tehnologije za pružanje značajne dobrobiti u procesu razvoja softvera.

Ključne riječi: ponovno korištenje, distribuirani objekti, komponenta, CORBA, DCOM.